US010175981B2

(12) **United States Patent**
Anderson et al.

(10) **Patent No.:** **US 10,175,981 B2**
(45) **Date of Patent:** **Jan. 8, 2019**

(54) METHOD TO CONTROL THE NUMBER OF ACTIVE VECTOR LANES FOR POWER EFFICIENCY

(71) Applicant: **Texas Instruments Incorporated**, Dallas, TX (US)

(72) Inventors: **Timothy David Anderson**, University Park, TX (US); **Duc Quang Bui**, Grand Prairie, TX (US)

(73) Assignee: **TEXAS INSTRUMENTS INCORPORATED**, Dallas, TX (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 216 days.

(21) Appl. No.: **14/326,928**

(22) Filed: **Jul. 9, 2014**

(65) **Prior Publication Data**

US 2015/0019896 A1 Jan. 15, 2015

Related U.S. Application Data

(60) Provisional application No. 61/844,124, filed on Jul. 9, 2013.

(51) **Int. Cl.**

G06F 1/32	(2006.01)
G06F 9/30	(2018.01)
G06F 9/38	(2018.01)

(52) **U.S. Cl.**

CPC ***G06F 9/30036*** (2013.01); ***G06F 1/3206*** (2013.01); ***G06F 1/3287*** (2013.01); ***G06F 9/3013*** (2013.01); ***G06F 9/30014*** (2013.01); ***G06F 9/30018*** (2013.01); ***G06F 9/30072*** (2013.01); ***G06F 9/30109*** (2013.01); ***G06F 9/30112*** (2013.01); ***G06F 9/3887*** (2013.01); *Y02D 10/171* (2018.01)

(58) **Field of Classification Search**

CPC G06F 1/3287

See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

6,839,828 B2 *	1/2005	Gschwind	G06F 9/30043	712/20
9,009,506 B2 *	4/2015	Fatemi	G06F 9/3836	713/320
2005/0044434 A1 *	2/2005	Kahle	G06F 1/3203	713/320
2006/0155964 A1 *	7/2006	Totsuka	G06F 9/30101	712/214
2006/0179329 A1 *	8/2006	Terechko	G06F 1/3203	713/300
2006/0282826 A1 *	12/2006	Dockser	G06F 1/206	717/127
2011/0047349 A1 *	2/2011	Hayashi	G06F 1/3203	712/22
2013/0268794 A1 *	10/2013	Hancock	G06F 7/57	713/323

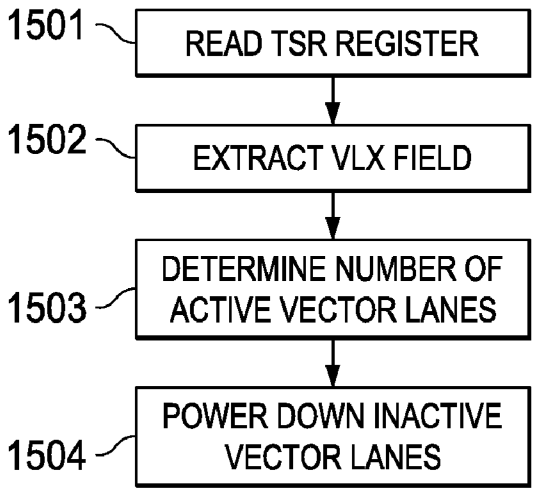
* cited by examiner

Primary Examiner — Robert A Cassity
Assistant Examiner — Gary Collins
(74) *Attorney, Agent, or Firm* — Kenneth Liu; Charles A. Brill; Frank D. Cimino

(57) **ABSTRACT**

The vector data path is divided into smaller vector lanes. The number of active vector lanes is controllable on the fly by the programmer to match the requirements of the executing program, and inactive vector lanes are powered down by the CPU to increase power efficiency of the vector processor.

15 Claims, 8 Drawing Sheets



1501
READ TSR REGISTER
1502
EXTRACT VLX FIELD
1503
DETERMINE NUMBER OF ACTIVE VECTOR LANES
1504
POWER DOWN INACTIVE VECTOR LANES

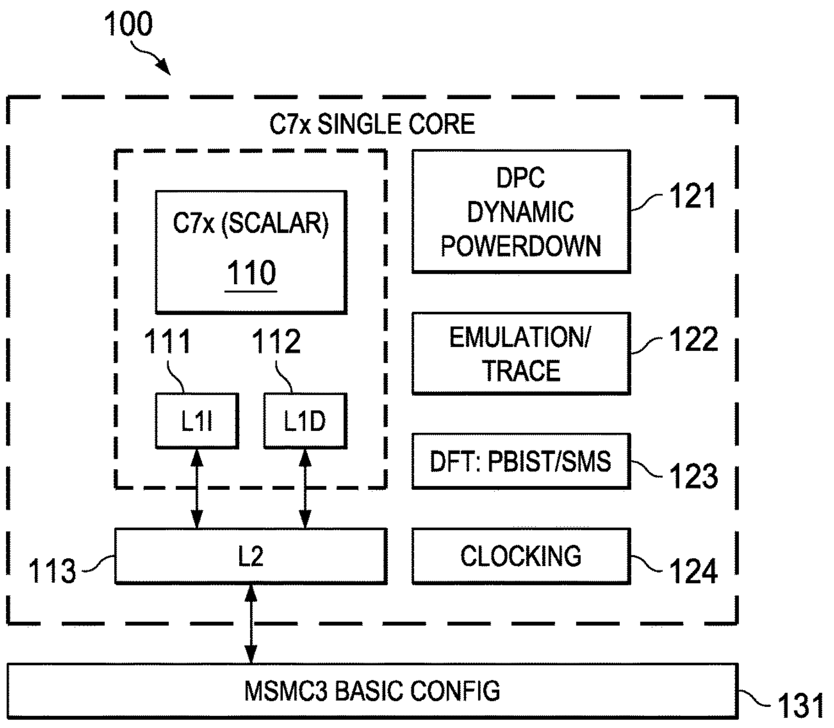

100
C7x SINGLE CORE
C7x (SCALAR)
110
111
112
L1I
L1D
L2
113
DPC
DYNAMIC
POWERDOWN
121
EMULATION/
TRACE
122
DFT: PBIST/SMS
123
CLOCKING
124
MSMC3 BASIC CONFIG
131

FIG. 1

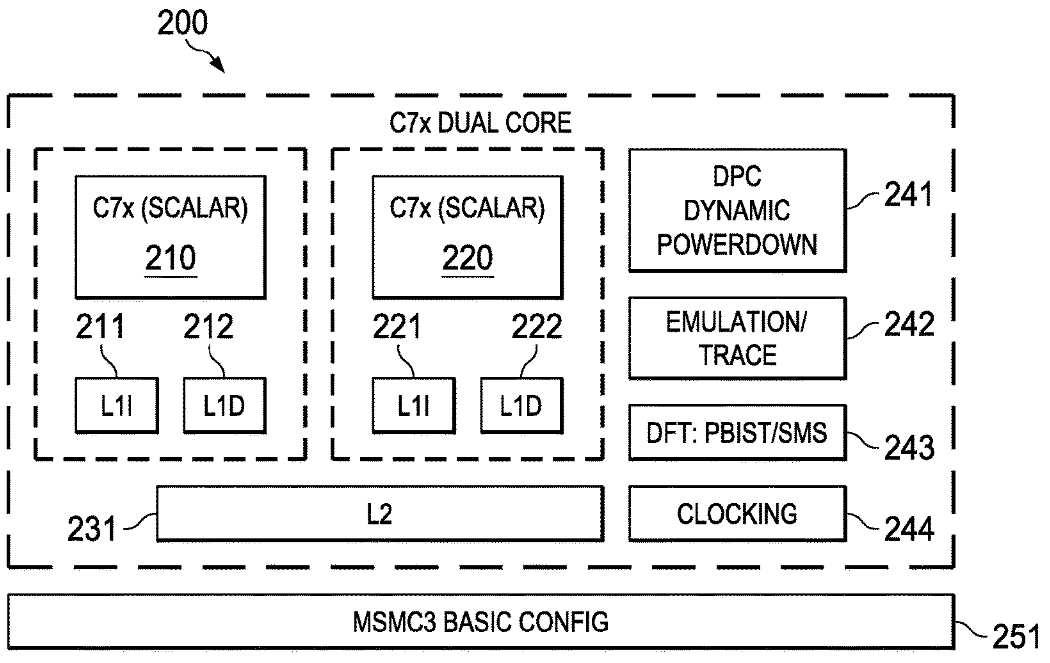

200
C7x DUAL CORE
C7x (SCALAR)
210
211
212
L1I
L1D
C7x (SCALAR)
220
221
222
L1I
L1D
L2
231
DPC
DYNAMIC
POWERDOWN
241
EMULATION/
TRACE
242
DFT: PBIST/SMS
243
CLOCKING
244
MSMC3 BASIC CONFIG
251

FIG. 2

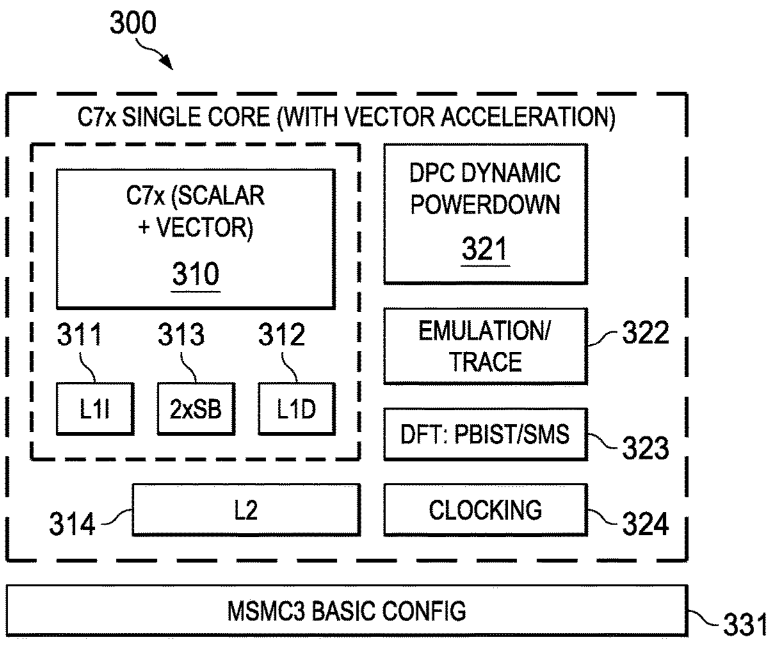

300
C7x SINGLE CORE (WITH VECTOR ACCELERATION)
C7x (SCALAR + VECTOR) 310
311
313
312
L1I
2xSB
L1D
314
L2
DPC DYNAMIC POWERDOWN 321
EMULATION/ TRACE
322
DFT: PBIST/SMS
323
CLOCKING
324
MSMC3 BASIC CONFIG
331

FIG. 3

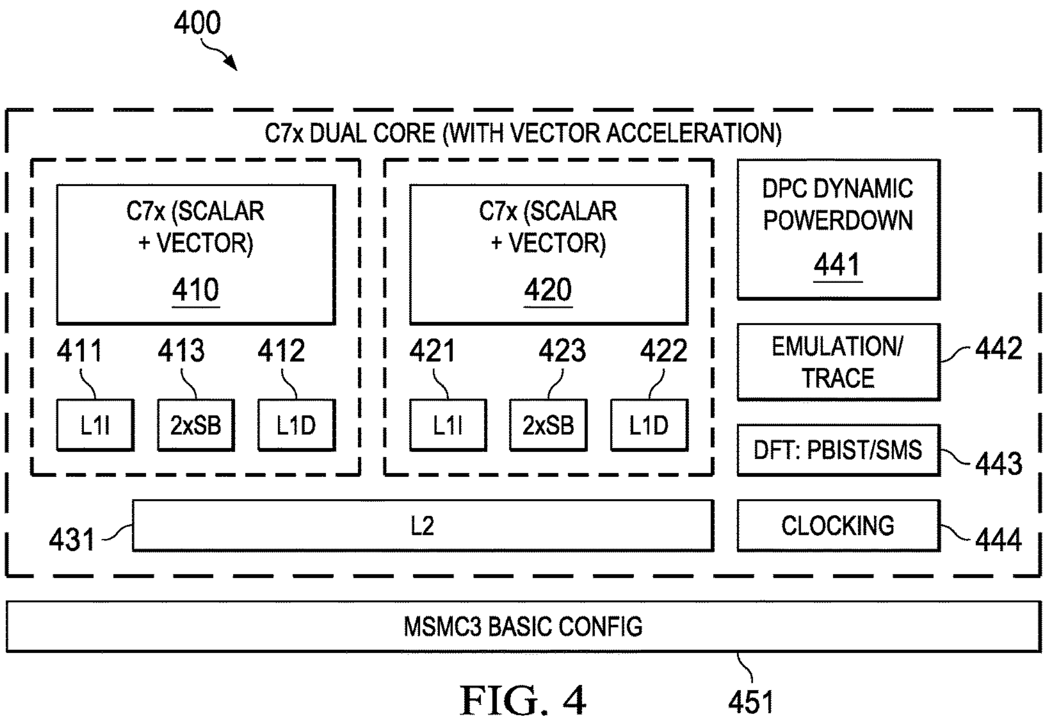

400
C7x DUAL CORE (WITH VECTOR ACCELERATION)
C7x (SCALAR + VECTOR) 410
411
413
412
L1I
2xSB
L1D
C7x (SCALAR + VECTOR) 420
421
423
422
L1I
2xSB
L1D
431
L2
DPC DYNAMIC POWERDOWN 441
EMULATION/ TRACE
442
DFT: PBIST/SMS
443
CLOCKING
444
MSMC3 BASIC CONFIG
451

FIG. 4

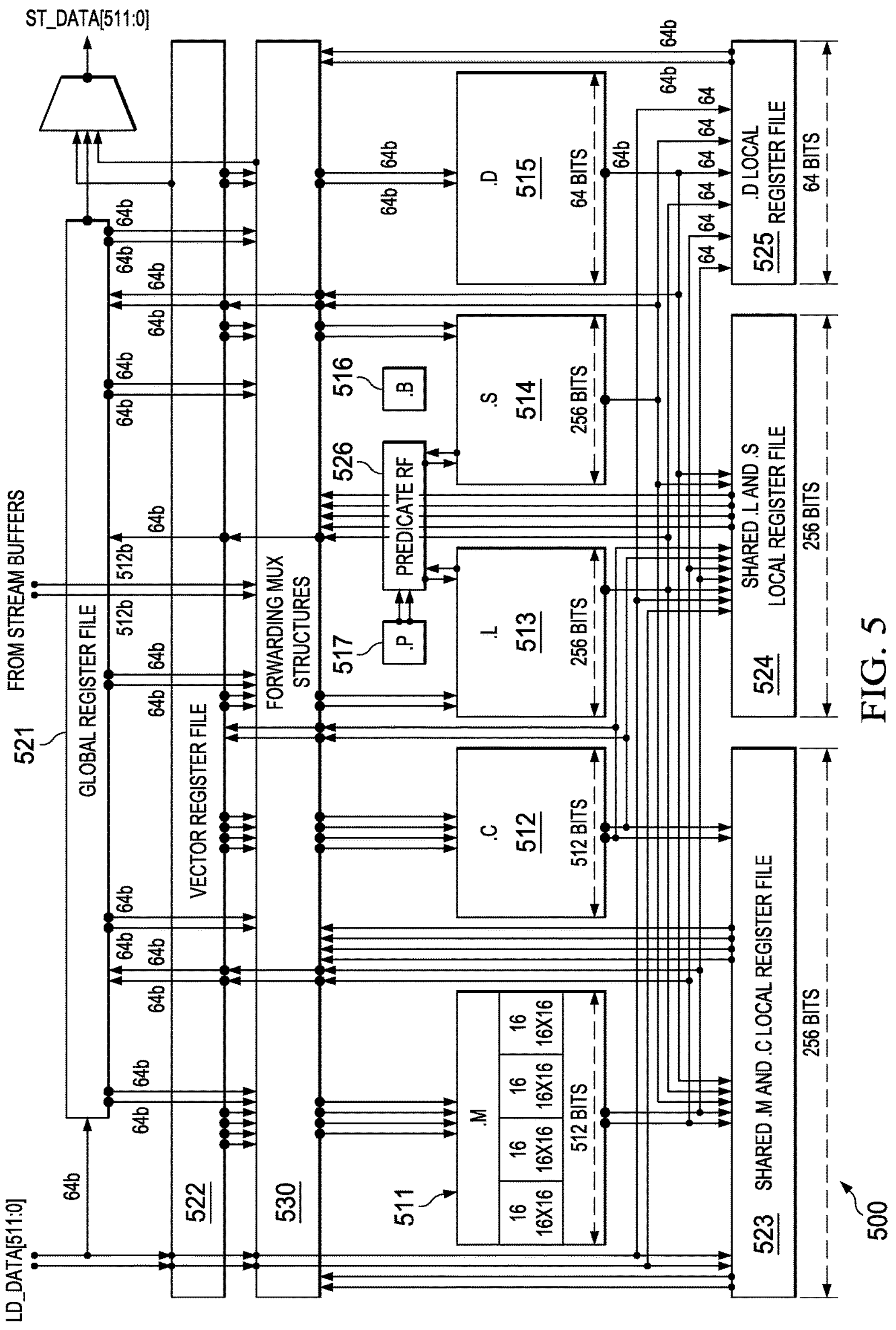

ST_DATA[511:0]
LD_DATA[511:0]
FROM STREAM BUFFERS
521
GLOBAL REGISTER FILE
VECTOR REGISTER FILE
522
FORWARDING MUX STRUCTURES
530
.M
511
16
16X16
512 BITS
.C
512
.L
513
256 BITS
.S
514
.D
515
64 BITS
.B
516
.P
517
PREDICATE RF
526
SHARED .M AND .C LOCAL REGISTER FILE
523
SHARED .L AND .S LOCAL REGISTER FILE
524
.D LOCAL REGISTER FILE
525
64b
512b
64
500

FIG. 5

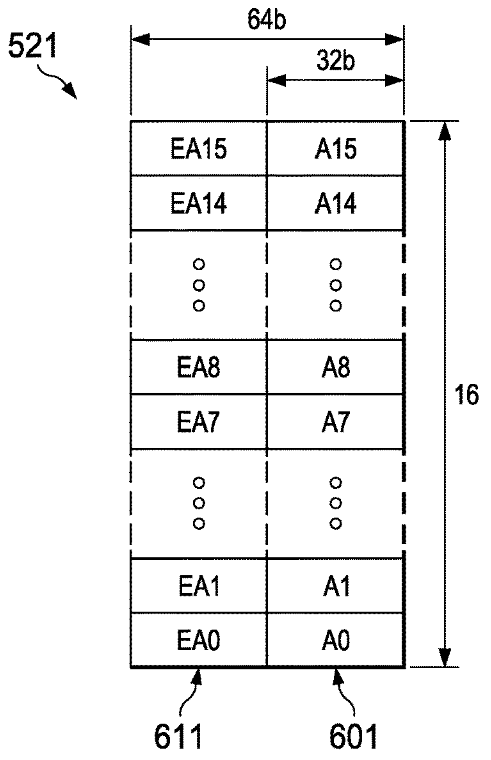

521
64b
32b
EA15
A15
EA14
A14
EA8
A8
EA7
A7
EA1
A1
EA0
A0
16
611
601

FIG. 6

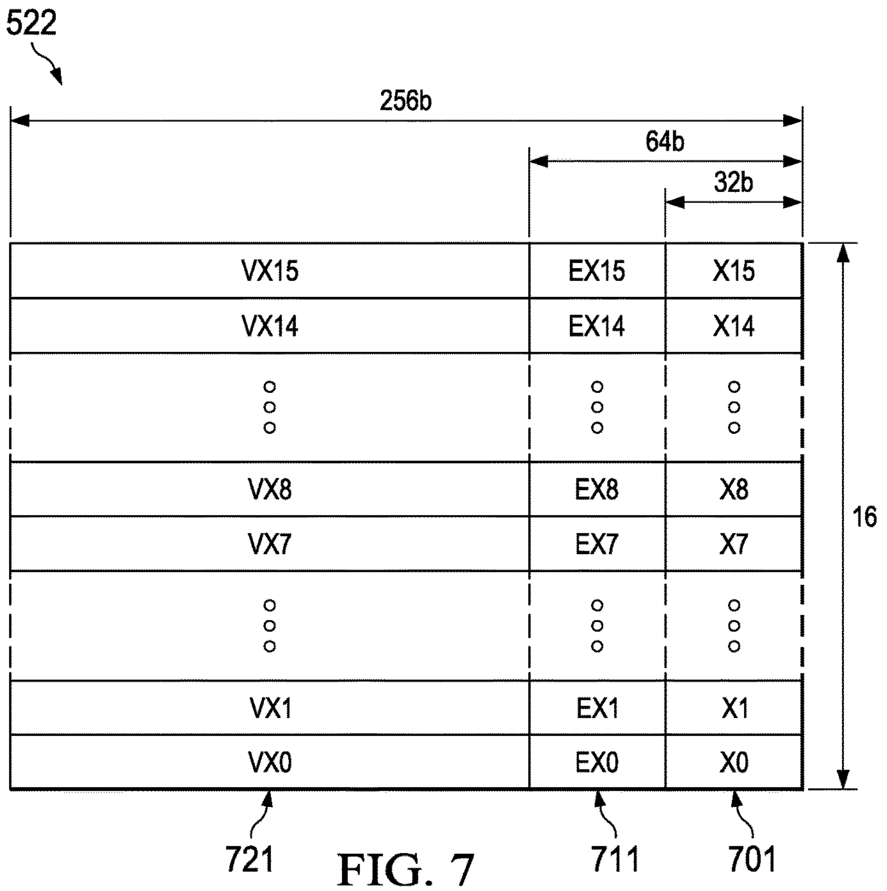

522
256b
64b
32b
VX15
EX15
X15
VX14
EX14
X14
VX8
EX8
X8
VX7
EX7
X7
VX1
EX1
X1
VX0
EX0
X0
16
721
711
701

FIG. 7

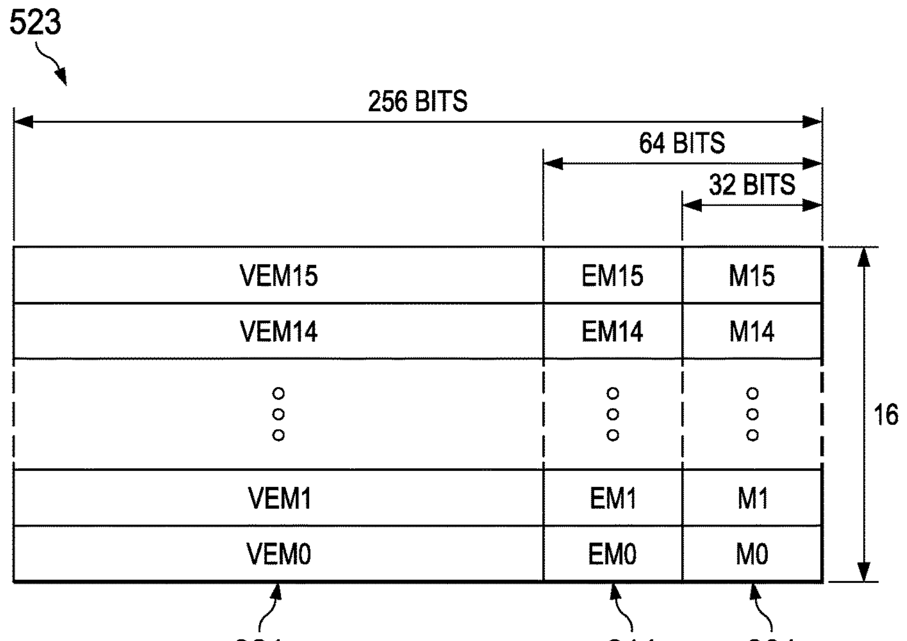


FIG. 8

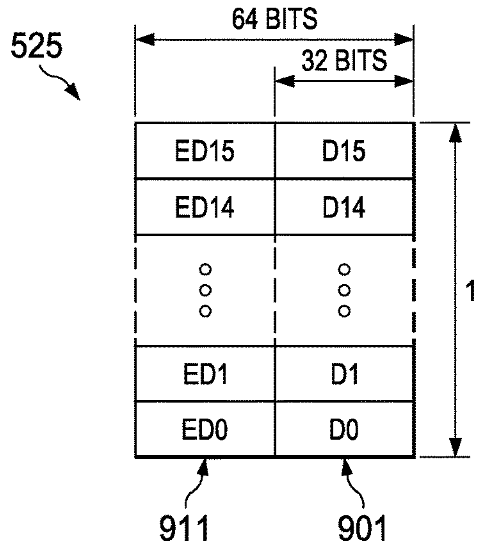


FIG. 9

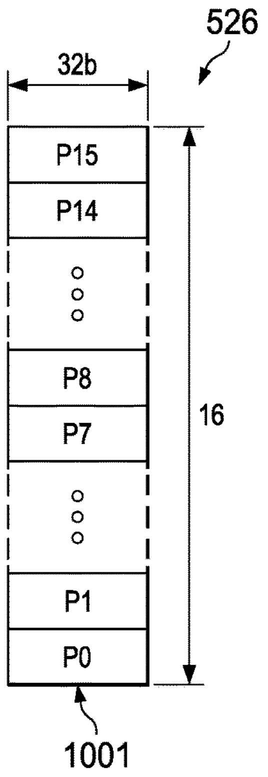


FIG. 10

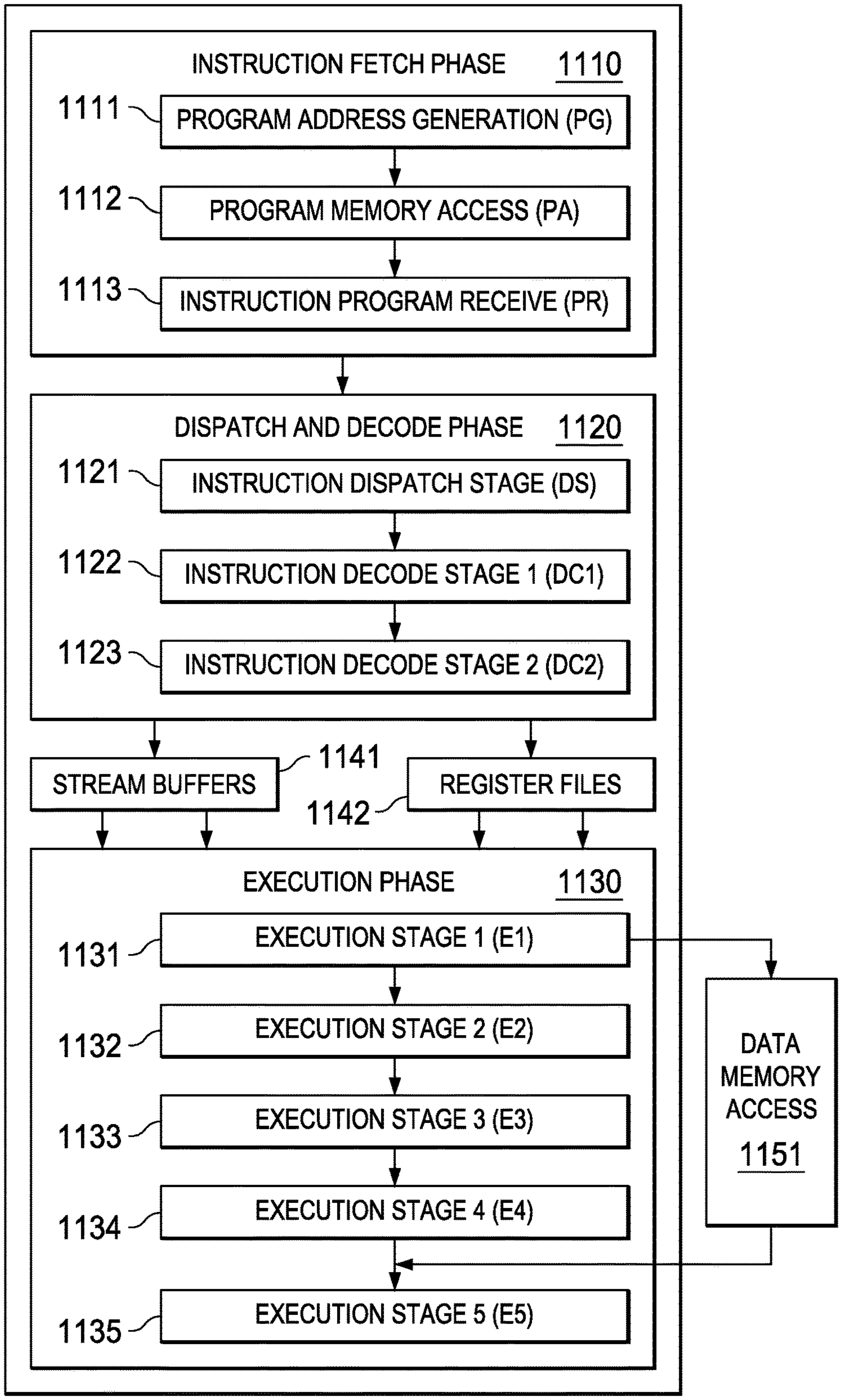

INSTRUCTION FETCH PHASE
1110
1111
PROGRAM ADDRESS GENERATION (PG)
1112
PROGRAM MEMORY ACCESS (PA)
1113
INSTRUCTION PROGRAM RECEIVE (PR)
DISPATCH AND DECODE PHASE
1120
1121
INSTRUCTION DISPATCH STAGE (DS)
1122
INSTRUCTION DECODE STAGE 1 (DC1)
1123
INSTRUCTION DECODE STAGE 2 (DC2)
STREAM BUFFERS
1141
REGISTER FILES
1142
EXECUTION PHASE
1130
1131
EXECUTION STAGE 1 (E1)
1132
EXECUTION STAGE 2 (E2)
1133
EXECUTION STAGE 3 (E3)
1134
EXECUTION STAGE 4 (E4)
1135
EXECUTION STAGE 5 (E5)
DATA MEMORY ACCESS
1151

FIG. 11

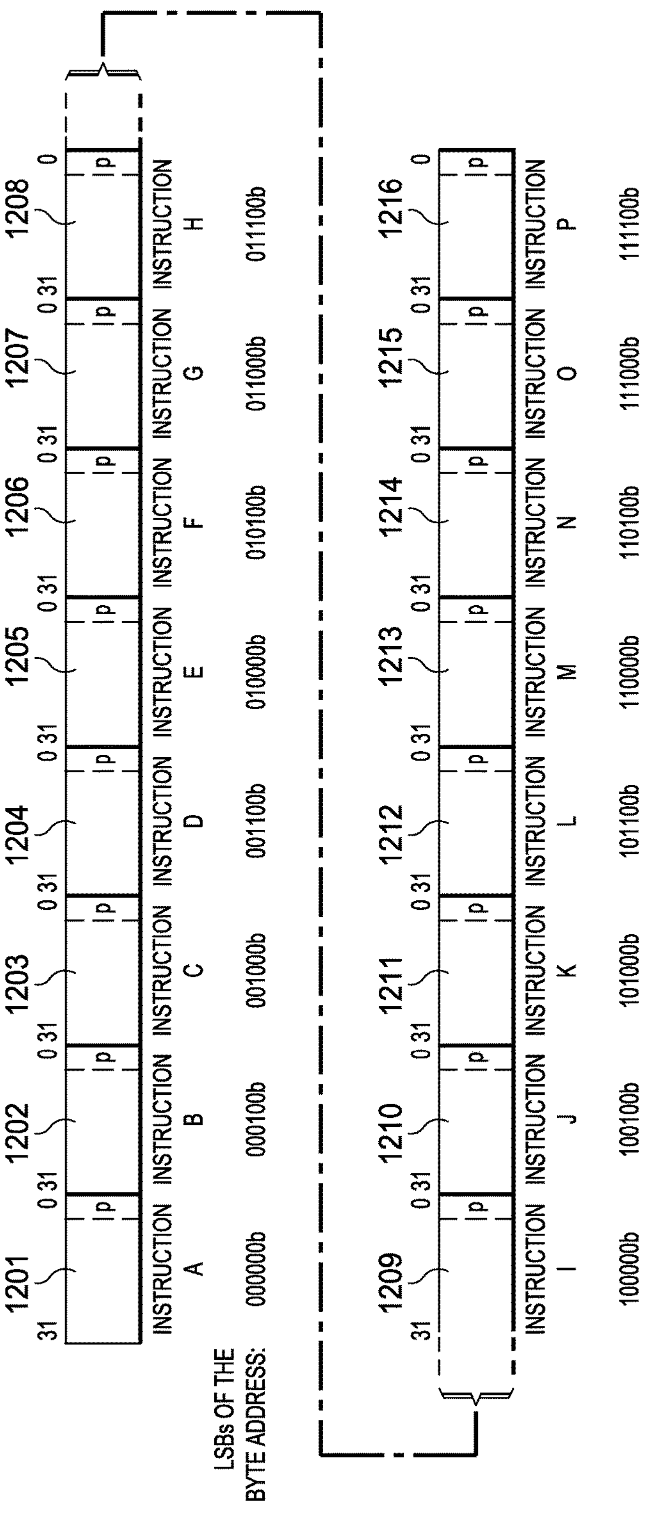

1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
31
0
p
INSTRUCTION A
INSTRUCTION B
INSTRUCTION C
INSTRUCTION D
INSTRUCTION E
INSTRUCTION F
INSTRUCTION G
INSTRUCTION H
INSTRUCTION I
INSTRUCTION J
INSTRUCTION K
INSTRUCTION L
INSTRUCTION M
INSTRUCTION N
INSTRUCTION O
INSTRUCTION P
LSBs OF THE BYTE ADDRESS:
000000b
000100b
001000b
001100b
010000b
010100b
011000b
011100b
100000b
100100b
101000b
101100b
110000b
110100b
111000b
111100b

FIG. 12

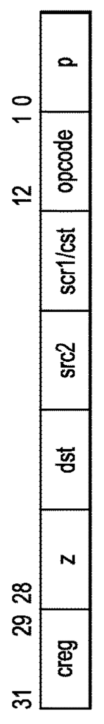

31
29 28
12
1 0
creg
z
dst
src2
scr1/cst
opcode
p

FIG. 13

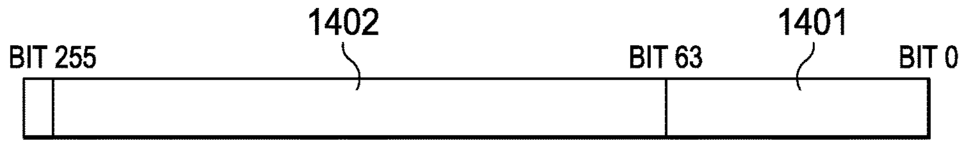

1402
1401
BIT 255
BIT 63
BIT 0

FIG. 14

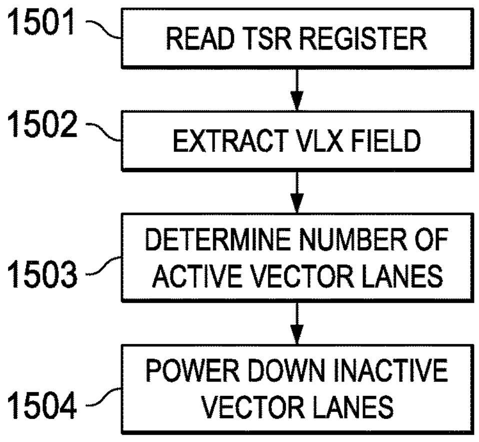

1501
READ TSR REGISTER
1502
EXTRACT VLX FIELD
1503
DETERMINE NUMBER OF ACTIVE VECTOR LANES
1504
POWER DOWN INACTIVE VECTOR LANES

FIG. 15

METHOD TO CONTROL THE NUMBER OF ACTIVE VECTOR LANES FOR POWER EFFICIENCY

CLAIM OF PRIORITY

This application claims priority under 35 U.S.C. 119(e)(1) to U.S. Provisional Application No. 61844124 filed 9 Jul. 2013.

TECHNICAL FIELD OF THE INVENTION

The technical field of this invention is digital data processing.

BACKGROUND OF THE INVENTION

Vector processors consume high amounts of power due to the wide data width. A normal vector unit can only be turned on or off in its entirety, hence wasting power if it only executes smaller data width operations. This invention specifies a method to turn on and off a portion of the vector data path on the fly.

SUMMARY OF THE INVENTION

The vector data path is divided into smaller vector lanes. For instance, a 256-bit wide vector data path can be divided into 4 smaller 64-bit vector lanes. The invention allows the programmer to control the number of active vector lanes within a vector data path by writing into a control register field the number of active vector lanes.

BRIEF DESCRIPTION OF THE DRAWINGS

These and other aspects of this invention are illustrated in the drawings, in which:

FIG. **1** illustrates a single core scalar processor according to one embodiment of this invention;

FIG. **2** illustrates a dual core scalar processor according to another embodiment of this invention;

FIG. **3** illustrates a single core vector processor according to a further embodiment of this invention;

FIG. **4** illustrates a dual core vector processor according to a further embodiment of this invention;

FIG. **5** illustrates construction of one embodiment of the CPU of this invention;

FIG. **6** illustrates the global scalar register file;

FIG. **7** illustrates global vector register file;

FIG. **8** illustrates the local vector register file shared by the multiply and correlation functional units;

FIG. **9** illustrates local register file of the load/store unit;

FIG. **10** illustrates the predicate register file;

FIG. **11** illustrates the pipeline phases of the central processing unit according to a preferred embodiment of this invention;

FIG. **12** illustrates sixteen instructions of a single fetch packet;

FIG. **13** illustrates an example of the instruction coding of instructions used by this invention;

FIG. **14** illustrates a reduced width vector data path; and

FIG. **15** illustrates a flow chart of one embodiment of the invention.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS

FIG. **1** illustrates a single core scalar processor according to one embodiment of this invention. Single core processor

100 includes a scalar central processing unit (CPU) **110** coupled to separate level one instruction cache (L1I) **111** and level one data cache (L1D) **112**. Central processing unit core **110** could be constructed as known in the art and would typically include a register file, an integer arithmetic logic unit, an integer multiplier and program flow control units. Single core processor **100** includes a level two combined instruction/data cache (L2) **113** that holds both instructions and data. In the preferred embodiment scalar central processing unit (CPU) **110**, level one instruction cache (L1I) **111**, level one data cache (L1D) **112** and level two combined instruction/data cache (L2) **113** are formed on a single integrated circuit.

In a preferred embodiment this single integrated circuit also includes auxiliary circuits such as power control circuit **121**, emulation/trace circuits **122**, design for test (DST) programmable built-in self test (PBIST) circuit **123** and clocking circuit **124**. External to CPU **110** and possibly integrated on single integrated circuit **100** is memory controller **131**.

CPU **110** operates under program control to perform data processing operations upon defined data. The program controlling CPU **110** consists of a plurality of instructions that must be fetched before decoding and execution. Single core processor **100** includes a number of cache memories. FIG. **1** illustrates a pair of first level caches. Level one instruction cache (L1I) **111** stores instructions used by CPU **110**. CPU **110** first attempts to access any instruction from level one instruction cache **121**. Level one data cache (L1D) **112** stores data used by CPU **110**. CPU **110** first attempts to access any required data from level one data cache **112**. The two level one caches (L1I **111** and L1D **112**) are backed by a level two unified cache (L2) **113**. In the event of a cache miss to level one instruction cache **111** or to level one data cache **112**, the requested instruction or data is sought from level two unified cache **113**. If the requested instruction or data is stored in level two unified cache **113**, then it is supplied to the requesting level one cache for supply to central processing unit core **110**. As is known in the art, the requested instruction or data may be simultaneously supplied to both the requesting cache and CPU **110** to speed use.

Level two unified cache **113** is further coupled to higher level memory systems via memory controller **131**. Memory controller **131** handles cache misses in level two unified cache **113** by accessing external memory (not shown in FIG. **1**). Memory controller **131** handles all memory centric functions such as cacheabilty determination, error detection and correction, address translation and the like. Single core processor **100** may be a part of a multiprocessor system. In that case memory controller **131** handles data transfer between processors and maintains cache coherence among processors.

FIG. **2** illustrates a dual core processor according to another embodiment of this invention. Dual core processor **200** includes first CPU **210** coupled to separate level one instruction cache (L1I) **211** and level one data cache (L1D) **212** and second CPU **220** coupled to separate level one instruction cache (L1I) **221** and level one data cache (L1D) **212**. Central processing units **210** and **220** are preferably constructed similar to CPU **110** illustrated in FIG. **1**. Dual core processor **200** includes a single shared level two combined instruction/data cache (L2) **231** supporting all four level one caches (L1I **211**, L1D **212**, L1I **221** and L1D **222**). In the preferred embodiment CPU **210**, level one instruction cache (L1I) **211**, level one data cache (L1D) **212**, CPU **220**, level one instruction cache (L1I) **221**, level one data cache (L1D) **222** and level two combined instruction/

data cache (L2) **231** are formed on a single integrated circuit. This single integrated circuit preferably also includes auxiliary circuits such as power control circuit **241**, emulation/trace circuits **242**, design for test (DST) programmable built-in self test (PBIST) circuit **243** and clocking circuit **244**. This single integrated circuit may also include memory controller **251**.

FIGS. **3** and **4** illustrate single core and dual core processors similar to that shown respectively in FIGS. **1** and **2**. FIGS. **3** and **4** differ from FIGS. **1** and **2** in showing vector central processing units. As further described below Single core vector processor **300** includes a vector CPU **310**. Dual core vector processor **400** includes two vector CPUs **410** and **420**. Vector CPUs **310**, **410** and **420** include wider data path operational units and wider data registers than the corresponding scalar CPUs **110**, **210** and **220**.

Vector CPUs **310**, **410** and **420** further differ from the corresponding scalar CPUs **110**, **210** and **220** in the inclusion of streaming engine **313** (FIG. **3**) and streaming engines **413** and **423** (FIG. **5**). Streaming engines **313**, **413** and **423** are similar. Streaming engine **313** transfers data from level two unified cache **313** (L2) to a vector CPU **310**. Streaming engine **413** transfers data from level two unified cache **431** to vector CPU **410**. Streaming engine **423** transfers data from level two unified cache **431** to vector CPU **420**. In accordance with the preferred embodiment each streaming engine **313**, **413** and **423** manages up to two data streams.

Each streaming engine **313**, **413** and **423** transfer data in certain restricted circumstances. A stream consists of a sequence of elements of a particular type. Programs that operate on streams read the data sequentially, operating on each element in turn. Every stream has the following basic properties. The stream data have a well-defined beginning and ending in time. The stream data have fixed element size and type throughout the stream. The stream data have fixed sequence of elements. Thus programs cannot seek randomly within the stream. The stream data is read-only while active. Programs cannot write to a stream while simultaneously reading from it. Once a stream is opened the streaming engine: calculates the address; fetches the defined data type from level two unified cache; performs data type manipulation such as zero extension, sign extension, data element sorting/swapping such as matrix transposition; and delivers the data directly to the programmed execution unit within the CPU. Streaming engines are thus useful for real-time digital filtering operations on well-behaved data. Streaming engines free these memory fetch tasks from the corresponding CPU enabling other processing functions.

The streaming engines provide the following benefits. The permit multi-dimensional memory accesses. They increase the available bandwidth to the functional units. They minimize the number of cache miss stalls since the stream buffer can bypass L1D cache. They reduce the number of scalar operations required in the loop to maintain. They manage the address pointers. They handle address generation automatically freeing up the address generation instruction slots and the .D unit for other computations.

FIG. **5** illustrates construction of one embodiment of the CPU of this invention. Except where noted this description covers both scalar CPUs and vector CPUs. The CPU of this invention includes plural execution units multiply unit **511** (.M), correlation unit **512** (.C), arithmetic unit **513** (.L), arithmetic unit **514** (.S), load/store unit **515** (.D), branch unit **516** (.B) and predication unit **517** (.P). The operation and relationships of these execution units are detailed below.

Multiply unit **511** primarily performs multiplications. Multiply unit **511** accepts up to two double vector operands and produces up to one double vector result. Multiply unit **511** is instruction configurable to perform the following operations: various integer multiply operations, with precision ranging from 8-bits to 64-bits; various regular and complex dot product operations; and various floating point multiply operations; bit-wise logical operations; moves; as well as adds and subtracts. As illustrated in FIG. **5** multiply unit **511** includes hardware for four simultaneous 16 bit by 16 bit multiplications. Multiply unit **511** may access global scalar register file **521**, global vector register file **522** and shared .M and C. local register **523** file in a manner described below. Forwarding multiplexer **530** mediates the data transfer between global scalar register file **521**, global vector register file **522**, the corresponding streaming engine and multiply unit **511**.

Correlation unit **512** (.C) accepts up to two double vector operands and produces up to one double vector result. Correlation unit **512** supports these major operations. In support of WCDMA "Rake" and "Search" instructions correlation unit **512** performs up to 512 2-bit PN*8-bit I/Q complex multiplies per clock cycle. Correlation unit **512** performs 8-bit and 16-bit Sum-of-Absolute-Difference (SAD) calculations performing up to 512 SADs per clock cycle. Correlation unit **512** performs horizontal add and horizontal min/max instructions. Correlation unit **512** performs vector permutes instructions. Correlation unit **512** includes contains 8 256-bit wide control registers. These control registers are used to control the operations of certain correlation unit instructions. Correlation unit **512** may access global scalar register file **521**, global vector register file **522** and shared .M and C. local register file **523** in a manner described below. Forwarding multiplexer **530** mediates the data transfer between global scalar register file **521**, global vector register file **522**, the corresponding streaming engine and correlation unit **512**.

CPU **500** includes two arithmetic units: arithmetic unit **513** (.L) and arithmetic unit **514** (.S). Each arithmetic unit **513** and arithmetic unit **514** accepts up to two vector operands and produces one vector result. The compute units support these major operations. Arithmetic unit **513** and arithmetic unit **514** perform various single-instruction-multiple-data (SIMD) fixed point arithmetic operations with precision ranging from 8-bit to 64-bits. Arithmetic unit **513** and arithmetic unit **514** perform various compare and minimum/maximum instructions which write results directly to predicate register file **526** (further described below). Arithmetic unit **513** and arithmetic unit **514** perform various SIMD floating point arithmetic operations with precision ranging from half-precision (16-bits), single precision (32-bits) to double precision (64-bits). Arithmetic unit **513** and arithmetic unit **514** perform specialized instructions to speed up various algorithms and functions. Arithmetic unit **513** and arithmetic unit **514** may access global scalar register file **521**, global vector register file **522**, shared .L and S. local register file **524** and predicate register file **526** in a manner described below. Forwarding multiplexer **530** mediates the data transfer between global scalar register file **521**, global vector register file **522**, the corresponding streaming engine and arithmetic units **513** and **514**.

Load/store unit **515** (.D) is primarily used for address calculations. Load/store unit **515** is expanded to accept scalar operands up to 64-bits and produces scalar result up to 64-bits. Load/store unit **515** includes additional hardware to perform data manipulations such as swapping, pack and unpack on the load and store data to reduce workloads on the other units. Load/store unit **515** can send out one load or store request each clock cycle along with the 44-bit physical address to level one data cache (L1D). Load or store data width can be 32-bits, 64-bits, 256-bits or 512-bits. Load/store unit **515** supports these major operations: 64-bit SIMD arithmetic operations; 64-bit bit-wise logical operations; and scalar and vector load and store data manipulations. Load/store unit **515** preferably includes a micro-TLB (table look-aside buffer) block to perform address translation from a 48-bit virtual address to a 44-bit physical address. Load/store unit **515** may access global scalar register file **521**, global vector register file **522** and .D local register file **525** in a manner described below. Forwarding multiplexer **530** mediates the data transfer between global scalar register file **521**, global vector register file **522**, the corresponding streaming engine and load/store unit **515**.

Branch unit **516** (.B) calculates branch addresses, performs branch predictions, and alters control flows dependent on the outcome of the prediction.

Predication unit **517** (.P) is a small control unit which performs basic operations on vector predication registers. Predication unit **517** has direct access to the vector predication registers **526**. Predication unit **517** performs different bit operations on the predication registers such as AND, ANDN, OR, XOR, NOR, BITR, NEG, SET, BITCNT, RMBD, BIT Decimate and Expand, etc.

FIG. **6** illustrates global scalar register file **521**. There are 16 independent 64-bit wide scalar registers. Each register of global scalar register file **521** can be read as 32-bits scalar data (designated registers A**0** to A**15** **601**) or 64-bits of scalar data (designated registers EA**0** to EA**15** **611**).

However, writes are always 64-bit, zero-extended to fill up to 64-bits if needed. All scalar instructions of all functional units can read or write to global scalar register file **521**. The instruction type determines the data size. Global scalar register file **521** supports data types ranging in size from 8-bits through 64-bits. A vector instruction can also write to the 64-bit global scalar registers **521** with the upper 192 bit data of the vector discarded. A vector instruction can also read 64-bit data from the global scalar register file **511**. In this case the operand is zero-extended in the upper 192-bit to form an input vector.

FIG. **7** illustrates global vector register file **522**. There are 16 independent 256-bit wide vector registers. Each register of global vector register file **522** can be read as 32-bits scalar data (designated registers X**0** to X**15** **701**), 64-bits of scalar data (designated registers EX**0** to EX**15** **711**), 256-bit vector data (designated registers VX**0** to VX**15** **721**) or 512-bit double vector data (designated DVX**0** to DVX**7**, not illustrated). In the current embodiment only multiply unit **511** and correlation unit **512** may execute double vector instructions. All vector instructions of all functional units can read or write to global vector register file **522**. Any scalar instruction of any functional unit can also access the low 32 or 64 bits of a global vector register file **522** register for read or write. The instruction type determines the data size.

FIG. **8** illustrates local vector register file **523**. There are 16 independent 256 bit wide vector registers. Each register of local vector register file **523** can be read as 32-bits scalar data (designated registers M**0** to M**15** **801**), 64-bits of scalar data (designated registers EM**0** to EM**15** **811**), 256 bit vector data (designated registers VM**0** to VM**15** **821**) or 512 bit double vector data (designated DVM**0** to DVM**7**, not illustrated). In the current embodiment only multiply unit **511** and correlation unit **512** may execute double vector instructions. All vector instructions of all functional units can write to local vector register file **523**. Only instructions of multiply

unit **511** and correlation unit **512** may read from local vector register file **523**. The instruction type determines the data size.

Multiply unit **511** may operate upon double vectors (512-bit data). Multiply unit **511** may read double vector data from and write double vector data to global vector register file **521** and local vector register file **523**. Register designations DVXx and DVMx are mapped to global vector register file **521** and local vector register file **523** as follows.

TABLE 1

Instruction Designation	Register Accessed
DVX0	VX1:VX0
DVX1	VX3:VX2
DVX2	VX5:VX4
DVX3	VX7:VX6
DVX4	VX9:VX8
DVX5	VX11:VX10
DVX6	VX13:VX12
DVX7	VX15:VX14
DVM0	VM1:VM0
DVM1	VM3:VM2
DVM2	VM5:VM4
DVM3	VM7:VM6
DVM4	VM9:VM8
DVM5	VM11:VM10
DVM6	VM13:VM12
DVM7	VM15:VM14

Each double vector designation maps to a corresponding pair of adjacent vector registers in either global vector register **522** or local vector register **523**. Designations DVX**0** to DVX**7** map to global vector register **522**. Designations DVM**0** to DVM**7** map to local vector register **523**.

Local vector register file **524** is similar to local vector register file **523**. There are 16 independent 256 bit wide vector registers. Each register of local vector register file **524** can be read as 32-bits scalar data (designated registers L**0** to L**15**), 64-bits of scalar data (designated registers EL**0** to EL**15**) or 256 bit vector data (designated registers VL**0** to VL**15**). All vector instructions of all functional units can write to local vector register file **524**. Only instructions of arithmetic unit **513** and arithmetic unit **514** may read from local vector register file **524**.

FIG. **9** illustrates local register file **525**. There are 16 independent 64 bit wide registers. Each register of local register file **525** can be read as 32-bits scalar data (designated registers DO to D**15** **901**) or 64-bits of scalar data (designated registers ED**0** to ED**15** **911**). All scalar and vector instructions of all functional units can write to local register file **525**. Only instructions of load/store unit **515** may read from local register file **525**. Any vector instructions can also write 64 bit data to local register file **525** with the upper 192 bit data of the result vector discarded. Any vector instructions can also read 64-bit data from the 64-bit local register file **525** registers. The return data is zero-extended in the upper 192-bit to form an input vector. The registers of local register file **525** can only be used as addresses in load/store instructions, not as store data or as sources for 64 bit arithmetic and logical instructions of load/store unit **515**.

FIG. **10** illustrates the predicate register file **517**. There are sixteen registers 32-bit registers in predicate register file **517**. Predicate register file **517** contains the results from vector comparison operations and is used by vector selection instructions and vector predicated store instructions. A small subset of special instructions can also read directly from predicate registers, performs operations and write back to a predicate register directly. There are also instructions which can transfer values between the global register files (**521** and **522**) and predicate register file **517**. Transfers between predicate register file **517** and local register files (**523**, **524** and **525**) are not supported. Each bit of a predication register (designated P**0** to P**15**) controls a byte of a vector data. Since a vector is 256-bits, the width of a predicate register equals 256/8=32 bits. The predicate register file can be written to by vector comparison operations to store the results of the vector compares.

A CPU such as CPU **110**, **210**, **220**, **310**, **410** or **420** operates on an instruction pipeline. This instruction pipeline can dispatch up to nine parallel 32-bits slots to provide instructions to the seven execution units (multiply unit **511**, correlation unit **512**, arithmetic unit **513**, arithmetic unit **514**, load/store unit **515**, branch unit **516** and predication unit **517**) every cycle. Instructions are fetched instruction packets of fixed length further described below. All instructions require the same number of pipeline phases for fetch and decode, but require a varying number of execute phases.

FIG. **11** illustrates the following pipeline phases: program fetch phase **1110**, dispatch and decode phases **1120** and execution phases **1130**. Program fetch phase **1110** includes three stages for all instructions. Dispatch and decode phases **1120** include three stages for all instructions. Execution phase **1130** includes one to four stages dependent on the instruction.

Fetch phase **1110** includes program address generation stage **1111** (PG), program access stage **1112** (PA) and program receive stage **1113** (PR). During program address generation stage **1111** (PG), the program address is generated in the CPU and the read request is sent to the memory controller for the level one instruction cache L1I. During the program access stage **1112** (PA) the level one instruction cache L1I processes the request, accesses the data in its memory and sends a fetch packet to the CPU boundary. During the program receive stage **1113** (PR) the CPU registers the fetch packet.

Instructions are always fetched sixteen words at a time. FIG. **12** illustrates this fetch packet. FIG. **12** illustrates 16 instructions **1201** to **1216** of a single fetch packet. Fetch packets are aligned on 512-bit (16-word) boundaries. The execution of the individual instructions is partially controlled by a p bit in each instruction. This p bit is preferably bit **0** of the instruction. The p bit determines whether the instruction executes in parallel with another instruction. The p bits are scanned from lower to higher address. If the p bit of an instruction is 1, then the next following instruction is executed in parallel with (in the same cycle as) that instruction I. If the p bit of an instruction is 0, then the next following instruction is executed in the cycle after the instruction. All instructions executing in parallel constitute an execute packet. An execute packet can contain up to nine instructions. Each instruction in an execute packet must use a different functional unit. An execute packet can contain up to nine 32-bit wide slots. A slot can either be a self-contained instruction or expand the constant field specified by the immediate preceding instruction. A slot can be used as conditional codes to apply to the instructions within the same fetch packet. A fetch packet can contain up to 2 constant extension slots and one condition code extension slot.

There are up to 11 distinct instruction slots, but scheduling restrictions limit to 9 the maximum number of parallel slots. The maximum nine slots are shared as follows: multiply unit **511**; correlation unit **512**; arithmetic unit **513**; arithmetic unit **514**; load/store unit **515**; branch unit **516** shared with predicate unit **517**; a first constant extension; a second constant extension; and a unitless instruction shared with a condition code extension. The last instruction in an execute packet has a p bit equal to 0.

The CPU and level one instruction cache L1I pipelines are de-coupled from each other. Fetch packet returns from level one instruction cache L1I can take different number of clock cycles, depending on external circumstances such as whether there is a hit in level one instruction cache L1I. Therefore program access stage **1112** (PA) can take several clock cycles instead of 1 clock cycle as in the other stages.

Dispatch and decode phases **1120** include instruction dispatch to appropriate execution unit stage **1121** (DS), instruction pre-decode stage **1122** (DC**1**); and instruction decode, operand reads stage **1123** (DC**2**). During instruction dispatch to appropriate execution unit stage **1121** (DS) the fetch packets are split into execute packets and assigned to the appropriate functional units. During the instruction pre-decode stage **1122** (DC**1**) the source registers, destination registers, and associated paths are decoded for the execution of the instructions in the functional units. During the instruction decode, operand reads stage **1123** (DC**2**) more detail unit decodes are done, as well as reading operands from the register files.

Execution phases **1130** includes execution stages **1131** to **1135** (E**1** to E**5**). Different types of instructions require different numbers of these stages to complete their execution. These stages of the pipeline play an important role in understanding the device state at CPU cycle boundaries.

During execute 1 stage **1131** (E**1**) the conditions for the instructions are evaluated and operands are operated on. As illustrated in FIG. **11**, execute 1 stage **1131** may receive operands from a stream buffer **1141** and one of the register files shown schematically as **1142**. For load and store instructions, address generation is performed and address modifications are written to a register file. For branch instructions, branch fetch packet in PG phase is affected. As illustrated in FIG. **11**, load and store instructions access memory here shown schematically as memory **1151**. For single-cycle instructions, results are written to a destination register file. This assumes that any conditions for the instructions are evaluated as true. If a condition is evaluated as false, the instruction does not write any results or have any pipeline operation after execute 1 stage **1131**.

During execute 2 stage **1132** (E**2**) load instructions send the address to memory. Store instructions send the address and data to memory. Single-cycle instructions that saturate results set the SAT bit in the control status register (CSR) if saturation occurs. For 2-cycle instructions, results are written to a destination register file.

During execute 3 stage **1133** (E**3**) data memory accesses are performed. Any multiply instructions that saturate results set the SAT bit in the control status register (CSR) if saturation occurs. For 3-cycle instructions, results are written to a destination register file.

During execute 4 stage **1134** (E**4**) load instructions bring data to the CPU boundary. For 4-cycle instructions, results are written to a destination register file.

During execute 5 stage **1135** (E**5**) load instructions write data into a register. This is illustrated schematically in FIG. **11** with input from memory **1151** to execute 5 stage **1135**.

FIG. **13** illustrates an example of the instruction coding of instructions used by this invention. Each instruction consists of 32 bits and controls the operation of one of the individually controllable functional units (multiply unit **511**, correlation unit **512**, arithmetic unit **513**, arithmetic unit **514**, load/store unit **515**). The bit fields are defined as follows.

The creg field and the z bit are optional fields used in conditional instructions. These bits are used for conditional instructions to identify the predicate register and the condition. The z bit (bit **28**) indicates whether the predication is based upon zero or not zero in the predicate register. If z=1, the test is for equality with zero. If z=0, the test is for nonzero. The case of creg=0 and z=0 is treated as always true to allow unconditional instruction execution. The creg field and the z field are encoded in the instruction as shown in Table 2.

TABLE 2

	Conditional Register			
	creg			z
	31	30	29	28
Unconditional	0	0	0	0
Reserved	0	0	0	1
B0	0	0	1	z
B1	0	1	0	z
B2	0	1	1	z
A1	1	0	0	z
A2	1	0	1	z
A0	1	1	0	z
Reserved	1	1	1	x

Note that "z" in the z bit column refers to the zero/not zero comparison selection noted above and "x" is a don't care state. This coding can only specify a subset of the 16 global scalar registers as predicate registers. This selection was made to preserve bits in the instruction coding. Note that unconditional instructions do not have these optional bits. For unconditional instructions these bits (**28** to **31**) are preferably used as additional opcode bits. However, if needed, an execute packet can contain a unique 32-bit condition code extension slot which contains the 4-bit CREGZ fields for the instructions which are in the same execute packet. Table 3 shows the coding of such a condition code extension slot.

TABLE 3

Bits	Functional Unit
3:0	.L
7:4	.S
11:5	.D
15:12	.M
19:16	.C
23:20	.B
28:24	Reserved
31:29	Reserved

Thus the condition code extension slot specifies bits decoded in the same way the creg/z bits assigned to a particular functional unit in the same execute packet.

The dst field specifies a register in a corresponding register file as the destination of the instruction results.

The scr2 field specifies a register in a corresponding register file as the second source operand.

The scr1/cst field has several meanings depending on the instruction opcode field (bits **2** to **12** and additionally bits **28** to **31** for unconditional instructions). The first meaning specifies a register of a corresponding register file as the first operand. The second meaning is an immediate constant. Depending on the instruction type, this is treated as an unsigned integer and zero extended to a specified data length or is treated as a signed integer and sign extended to the specified data length.

The opcode field (bits **2** to **12** for all instructions and additionally bits **28** to **31** for unconditional instructions) specifies the type of instruction and designates appropriate instruction options. This includes designation of the functional unit and operation performed. A detailed explanation of the opcode is beyond the scope of this invention except for the instruction options detailed below.

The p bit (bit **0**) marks the execute packets. The p-bit determines whether the instruction executes in parallel with the following instruction. The p-bits are scanned from lower to higher address. If p=1 for the current instruction, then the next instruction executes in parallel with the current instruction. If p=0 for the current instruction, then the next instruction executes in the cycle after the current instruction. All instructions executing in parallel constitute an execute packet. An execute packet can contain up to eight instructions. Each instruction in an execute packet must use a different functional unit.

Arithmetic units doing vector calculations operate on very wide data words, resulting in high power consumption. As an example, one embodiment shown in FIG. **5** implements 256 bit wide arithmetic units **513** and **514**, and a 512 bit wide correlation unit **512**. Since not all operations are this wide, power usage efficiency suffers when a 256 bit wide arithmetic unit is used to operate on 64 bit wide operands.

Power consumption may be minimized by dividing the vector data path width into smaller vector lanes. For example, as shown in FIG. **14** a 256 bit vector data path (vector lane) may be divided into a smaller 64 bit vector lane **1401** and a larger vector lane **1402**. If only the 64 bit vector lane **1401** is used, the larger vector lane **1402** may be powered off on the fly by the programmer. In one embodiment of this invention shown in FIG. **15**, the number of active vector lanes may be written into a control register (VLX field in the TSR register) in the Texas Instruments C7x CPU. **1501** reads this register, and the VLX field is extracted in **1502**. **1503** then determines the active vector lanes in use, and **1504** then powers down the unused vector lanes.

What is claimed is:

1. A data processor comprising:

a first execution unit;

a second execution unit;

a first local register file having a plurality of local registers;

an n-bit wide data path divided into a plurality of lanes, the plurality of lanes including a first lane having a size of x bits and a second lane having a size of y bits, wherein x and y are unequal integers, wherein the n-bit wide data path couples the first and second execution units to the first local register file, wherein the first and second execution units are both permitted to write to the local registers of the first local register file using the n-bit wide data path, the first execution unit is permitted to read from the local registers of the first local register file using the n-bit wide data path, and the second execution is not permitted to read from the local registers of the first local register file using the n-bit wide data path;

a control register configured to store control data indicating which of the plurality of lanes are active lanes for transfer of data over the n-bit wide data path in response to execution of a first instruction by the first execution unit of the data processor; and

a power control circuit that powers down lanes not indicated by the control data as an active lane.

2. The data processor of claim **1**, wherein y is greater than x.

3. The data processor of claim **2**, wherein y is equal to at least two times x.

4. The data processor of claim **2**, wherein y is a multiple of x.

5. The data processor of claim **4**, wherein y is equal to three times x.

6. The data processor of claim **5**, wherein x+y=n.

7. The data processor of claim **5**, wherein x+y<n, and plurality of lanes includes a third lane having a size equal to or less than n−(x+y).

8. The data processor of claim **1**, wherein:

the processor is a vector processor configured to execute vector instructions and scalar instructions; and

the n-bit wide data path is a vector data path.

9. The data processor of claim **1**, wherein each of the x-bit wide first lane and the y-bit wide second lane are contiguous, and wherein each of the x-bit wide first lane and the y-bit wide second lane are configured so that:

no portion of the x-bit wide first lane can be powered on without powering on the entire x-bit wide first lane or powered off without powering off the entire x-bit wide first lane; and

no portion of the y-bit wide second lane can be powered on without powering on the entire y-bit wide second lane or powered off without powering off the entire y-bit wide second lane.

10. The data processor of claim **1**, wherein:

n is equal to 256;

x is equal to 64; and

y is equal to 192.

11. The data processor of claim **1**, wherein the control data is stored in a predetermined bit field within the control register.

12. The data processor of claim **11**, wherein the control register includes a plurality of bits and the predetermined bit field is a subset of the plurality of bits.

13. The data processor of claim **1**, wherein the transfer of data over the n-bit wide data path in response to execution of the first instruction by the first execution unit is a read operation in which data stored in one of the local registers of the first local register file is read by the first execution unit.

14. The data processor of claim **1**, wherein the transfer of data over the n-bit wide data path in response to execution of the first instruction by the first execution unit is a write operation in which data provided the first execution unit is written to one of the local registers of the first local register file.

15. The data processor of claim **1**, wherein:

the first execution unit is an arithmetic unit; and

the second execution unit is one of a multiply unit or a correlation unit.

* * * * *